Read Me

Thank you for downloading!

If you like this asset then please leave a review/rating on the Asset Store, it really helps me out!

If you have any questions, feel free to email me at: carlos.wilkes@gmail.com

What is Lean Touch?

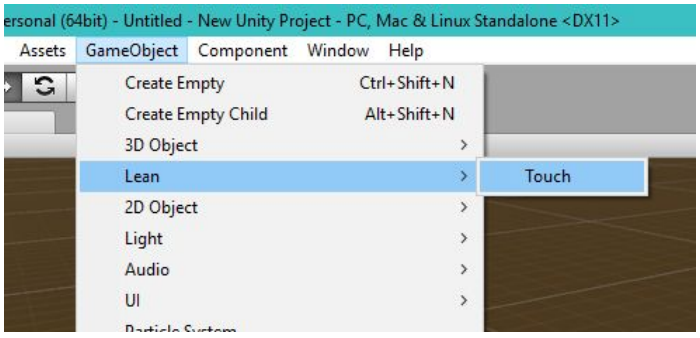
When you create mobile games you often want to make use of multi touch gestures, like pinch and twist. However, Unity makes this difficult to do, because they only provide the Input.touches array, requiring you to do all the calculations yourself.

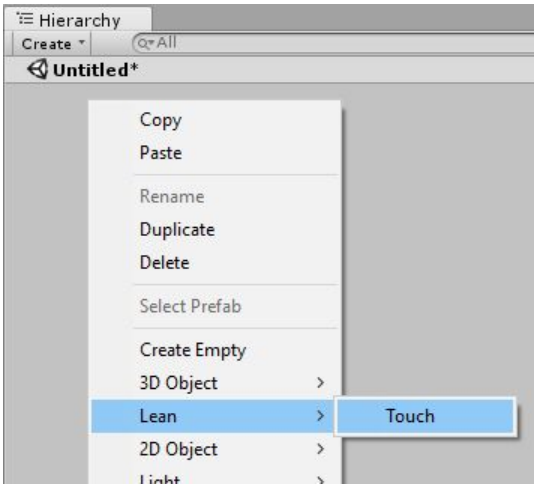
With LeanTouch you no longer have to worry about these issues, because all the touch gesture calculations are done for you in a very simple and elegant way.

LeanTouch also allows you to simulate multi touch gestures on desktop, so you don't have to waste lots of time deploying to your mobile devices while you setup your input.

How do I add Lean Touch to my game?

Click GameObject / Lean / Touch, or right click your Hierarchy window and go to Lean / Touch.





You should now see a new GameObject called 'LeanTouch' with the LeanTouch component selected.

When you enter play mode, this component will automatically convert all mouse and touch input into an easy to use format.

Remember that scripts using Lean Touch only work when there is a LeanTouch component active in your scene, so make sure to add one to every scene (or carry it over with DontDestroyOnLoad).

How do I use Lean Touch without code?

Lean Touch comes with many example components to do common tasks.

For example, if you want to spawn a prefab when a finger touches the screen, you can begin by making a new GameObject, and adding the LeanSpawnAt component. Inside this component, you'll see it has the 'Prefab' setting. You can browse or drag and drop your desired prefab here. Next, you can add the LeanFingerTap component. Inside this component, you'll see it has the 'OnTap' event. To link this event to the prefab spawning, you need to click the plus '+' button below where it says 'List is Empty', and in the bottom left box that says 'None (Object)', you need to drag and drop the LeanSpawn component you added earlier. Next, you need to click the 'No Function' dropdown, and select the 'LeanSpawnAt -> Dynamic LeanFinger -> Spawn' function. You can now hit play, and tapping a finger (or clicking) on the screen will spawn your prefab.

There are many other components that work in similar ways that can be connected together using only the editor. To find out how these work I recommend you browse through the demo scenes and look at the GameObjects to see what's going on.

How do I use Lean Touch with C#?

You can access all finger data from the Lean.Touch.LeanTouch class.

The easiest way to begin is by hooking into the static events it exposes.

For example, if you want to perform an action when a finger touches the screen, you can hook into the Lean.Touch.LeanTouch.OnFingerDown event. This event gets called every time a finger begins touching the screen, and gets passed a Lean.Touch.LeanFinger instance as a parameter. It's recommended you hook into these events from OnEnable, and unhook from OnDisable.

```
public class MyCoolScript : MonoBehaviour
{
    void OnEnable()
    {
        Lean.Touch.LeanTouch.OnFingerDown += OnFingerDown;
    }

    void OnDisable()
    {
        Lean.Touch.LeanTouch.OnFingerDown -= OnFingerDown;
    }

    void OnFingerDown(Lean.Touch.LeanFinger finger)
    {
        Debug.Log("Finger " + finger.Index + " just began touching the screen!");
    }
}
```

To see what other events are available, I recommend you read the LeanTouch.cs, script and its comments.

Another way to access finger data is to poll it directly from the Lean.Touch.LeanTouch.Fingers static list. This list stores all fingers that are currently touching the screen, and you can use this data at any time (e.g. Update) to quickly handle input.

```
public class MyCoolScript : MonoBehaviour
{
    void Update()
    {
        var fingers = Lean.Touch.LeanTouch.Fingers;

        Debug.Log("There are currently " + fingers.Count + " fingers touching the screen");
    }
}
```

If you need to modify or exclude certain fingers then I highly recommend you use the Lean.Touch.LeanTouch.GetFingers(...) method, which wich has common filter options, and will return a temporary list that you can further filter without breaking LeanTouch.

How do I handle multi-finger gestures (e.g. Pinch, Twist) from C#?

The Lean.Touch.LeanGesture class makes this very easy.

For example, if you want to find how many degrees the fingers were twisted in the last frame, you can call the Lean.Touch.LeanGesture.GetTwistDegrees() method, which will automatically calculate it from all fingers. This method also has an overload that allows you to pass a specific list of fingers if you don't want to use them all.

To see what other gestures are available, I recommend you read the LeanGesture.cs script and its comments.

Why are my inputs delayed by one frame?

By default, all components in Unity have an execution order value of 0 (Default Time). This means messages like Update (where inputs are handled) can potentially be executed in any order between your scripts. This can be an issue if the main LeanTouch script executes after your gameplay scripts, as your gameplay scripts will be working on one-frame old data, which can be noticeable if your game is running at low FPS, or requires precision inputs. To fix this:

1 - go to Edit → Project Settings → Script Execution Order.
2 - Below the 'Default Time' box, click the '+' button, and select the 'Lean.Touch.LeanTouch' script.
3 - Change the execution order value (default 100) to something below 0 (e.g. -100), or drag it above 'Default Time' and any other scripts.
4 - Click Apply, and enjoy!

How do I stop my touch controls from going through my UI from C#?

If you hook into any of the Lean.Touch.LeanTouch.OnFinger… events, you will get a Lean.Touch.LeanFinger instance. This class has the IsOverGui and StartedOverGui values you can check.

If you're polling from Lean.Touch.LeanTouch.Fingers and want to quickly remove fingers that are touching the GUI, you can instead get the fingers from the Lean.Touch.LeanTouch.GetFingers method, which has a setting to quickly exclude these, as well as restrict it to a certain amount of fingers.

Why do I have to keep typing 'Lean.Touch.' before everything?

To improve organization, all Lean Touch classes are inside the Lean.Touch namespace.

If you don't like typing Lean.Touch. each time, then you can add the following code to the top of your script: using Lean.Touch;

You can now just call LeanTouch.PointOverGui(...) etc

What is Lean Touch+?

Lean Touch+ is the paid version of Lean Touch. It has the same features as Lean Touch, but comes with MANY more example scripts that show you how to implement features found in many modern games and applications.

You can find more information about it here: https://www.assetstore.unity3d.com/#!/content/72356

Can I request a new demo scene?

Yes, if you have an idea for a demo scene that doesn't come with LeanTouch or LeanTouch+ then please request it via e-mail above.

Just make sure your demo scene idea doesn't require another asset or library, because I can't include those in this package!

If I like your demo scene idea then I'll even send you a free copy of Lean Touch+

How does the Screen Depth inspector setting work?

Fingers touching the screen only have a 2D XY coordinate, but many touch interactions require calculating a 3D XYZ point (e.g. LeanSpawnAt to spawn a prefab in 3D). This inspector setting/class handles this conversion, and is flexible enough to support 2D games, 3D games, perspective 2D games, and much more.

Understanding exactly how this works and can be used in your games can be difficult, so I recommend you examine the demo scenes to see how they are set up.

The way it can work with all these different scene types is via the dropdown box, and associated settings.

Camera Distance - This setting calculates a position in front of the current camera at the finger position, where the Distance setting is the distance from the camera in world space the point is pushed away. This setting is suitable for normal 3D games.

Depth Intercept - This setting calculates a ray in front of the current camera at the finger position, and finds the point where this ray intercepts a plane lying on the XY plane with the specified Z position. This setting is suitable for normal 2D games. If you're using standard 2D settings then a Z value of 0 should be used, but this can be adjusted for specific situations, because 2D sprites in Unity can be placed at any Z position and still work.

Physics Raycast - This setting calculates a ray in front of the current camera at the finger position, and finds the point where this ray hits the physics scene. This setting is suitable for 3D games where you need to spawn something on an object or similar.

Plane Intercept - This setting works similar to Depth Intercept, but allows you to specify a custom plane that can point in any direction, and optionally allows you to constrain or snap the final values.

Why do I have to link up so many components?

In earlier versions of Lean Touch, most components were self-contained, and calculated their own fingers. This made setup very simple, as you only had to add one component to instantly get its full functionality, but this simplicity came at a cost of making the code more complex and rigid. For example, there used to be a component that allowed you to rotate a GameObject when you swiped a finger, but if you wanted this to work with the just finger dragging, then this component would have to be duplicated and modified for each type of custom input you wanted. Doing this for swiping or dragging isn't too difficult, but for more complex behaviours like multi tapping, or swipe no-release, it required a lot of complex code to be added to each unique behaviour.

To solve this problem, Lean Touch 1.5.0 separated most of these features, so now there is a simple component that only handles rotation, and then you have separate finger handling components like LeanFingerSet, LeanFingerSwipe, LeanFingerSwipeNoRelease, and these use events that can be hooked to the rotation component. You can then use these same finger handling components to feed input to other components, and compose the exact type of touch behaviour you want without modifying any code.